The Combined L1/L2 Trigger Simulator

Dylan Casey, Josh Kalk, Roger Moore,Dugan O'Neil, ...

September 28, 2000

Contents

1 General Introduction **2**

2 The Combined Simulation Framework(tsim_l1l2) **2**

2.1 The DataFlow Dispatcher 2

2.2 RCP Configuration 3

2.3 Running tsim_l1l2 3

2.4 Built-in Configurations and Examples 4

2.5 Inter-Package Communication 5

2.5.1 The DataBroadcast Class 5

2.5.2 What about IOGEN? MBT Channels?? 5

3 Configuration via CoorSim **5**

4 Sending Data to L3 (l1l2collector) **6**

5 Adding an L1 Processor **6**

5.1 Being Part of the DataFlow 6

5.1.1 The inputs Function 7

5.1.2 The outputs Function 7

5.1.3 The ready function 7

5.2 Example: Toy Version of L1CTT 7

5.3 Being part of tsim_l1l2 12

5.4 Places to look for help 14

6 The L1 Framework (l1frm) **14**

7 Adding an L2 Worker **14**

8 Global Tools and Filters **15**

1 General Introduction

This document is intended both as a users guide and a developers guide to the combined L1/L2 trigger simulation. The opening sections should give someone familiar with the D0 software environment enough information to run the simulator. More detailed information for developers follows in later sections.

2 The Combined Simulation Framework(tsim_l1l2)

tsim_l1l2 is the package which controls the running of the combined level 1 and level 2 simulation. It functions mainly as a repository of configuration information (eg. rcp's) and an area from which to run the simulation. The only code included in the package is either related to communication between various level 1 and level 2 packages (eg. the DataBroadcast class, spy modules) or is simply example code. The tsim_l1l2 package takes advantage of the DataFlow Dispatcher of the offline framework [1] (which is described in more detail in the following sections) to pass information between packages. This chapter will describe how to compile and run the simulation using the tsim_l1l2 package.

2.1 The DataFlow Dispatcher

The offline framework's DataFlow Dispatcher allows data to be moved through our simulation in much the same way that our hardware transports data online. Each package acts on the principle "I will run when I have a full set of inputs". Users of the DataFlow need only configure packages with a list of required inputs and outputs, the order of execution of the packages is determined automatically by the framework at runtime.

As an example of the intelligent package-ordering possible in this model consider the level 1 / level 2 calorimeter trigger system. This system contains a level 1 cal processor package, three level 2 cal preprocessor packages and a level 2 global processor package. In a DataFlow model each of these packages inherits from the DFPackage class and includes "inputs" and "outputs" methods in which lists of required inputs and outputs are defined. By defining the level 1 output as a required input to the level 2 preprocessors the framework will guarantee that the level 1 cal processor will run first. Similarly, by defining the outputs of the level 2 preprocessors as required inputs of global it will be guaranteed to run last. At initialization time the framework will check all input and output lists for logical inconsistencies (eg. required inputs which cannot be produced as outputs by any package) and will issue an error message if any are found.

The DataFlow Dispatcher allows the l1/l2 trigger simulation to be viewed as a black box. An event enters the box and a Raw Data Chunk exits the box on its way to level 3. The communication among l1 and l2 packages is only seen by packages within the DataFlow. It allows the entire l1/l2 simulation, composed of numerous packages, to be included as a single package (tsim_l1l2).

2.2 RCP Configuration

All rcp files required to run the simulation should be stored in the tsim_l1l2/rcp directory. This includes configuration for each level 1 processor, each level 2 worker, global, etc. Main program control is implemented in rcp files such as head_example.rcp. This file sets up the offline framework controller and lists packages which will be used by the framework. In this rcp the internal l1/l2 DataFlow network appears as a single package. An example of such a controller is shown below:

```
string PackageName = "Controller"
string Packages = "newevent configure tsim dump"
string interfaces = "generate decide l2configure process dump"
string Flow = "generate decide l2configure process dump"
RCP newevent = < tsim_l1l2 newevent >
RCP configure = <tsim_coor L2CoorMgr>
RCP tsim = < tsim_l1l2 tsim_example >
RCP dump = < tsim_l1l2 dumpevent >
```

In this example the "tsim" package refers to an RCP which configures the DataFlow network. An example rcp to configure such a DataFlow is shown below:

```
string PackageName = "Dataflow"
string Packages = ( l1example l2pp global_example collector)
RCP l1example = < tsim_l1l2 l1_example >
RCP l2pp = < tsim_l1l2 l2pp_example >
RCP global_example = < tsim_l1l2 l2_global_example >
RCP collector = < tsim_l1l2 l1l2_collector >
```

This example contains a single l1 processor, a single l2 preprocessor, a global worker and the collector which writes simulation information to the Raw Data Chunk.

2.3 Running tsim_l1l2

These instructions assume that you have never before used tsim_l1l2 or similar packages. They also assume that all necessary packages are included in the latest D0 software release and do not need to be recompiled by the user. Experienced users may wish to skip to step 5.

1. Recite the usual incantations:
   - setup D0RunII versionnumber
   - setup d0cvs
2. start a new release area:

- newrel -t versionnumber mydir

3. enter the release area and checkout tsim_l1l2:
   - cd mydir
   - addpkg tsim_l1l2
4. set this directory as your working area:
   - d0setwa
5. make the tsim_l1l2 executable:
   - gmake tsim_l1l2.bin
6. run the simulation:
   - cd tsim_l1l2/bin
   - ./Runme.sh

Runme.sh optionally takes 2 commandline arguements, the first specifies the controller rcp which will be used to configre the simulation, the second specifies any framework commandline arguements (eg. -config). By default Runme.sh will choose to run with the head.rcp configuration file. An example of a different configuration is:

```
Runme.sh head_example.rcp -config
```

which chooses the head_example controller and passes the framework a -config arguement.

In order to modify the configuration of individual packages used in the simulation the rcp files in the tsim_l1l2 rcp directory should be modified. For example, by default the simulator will read from a datafile named input.data, this can be changed by editing tsim_l1l2/rcp/ReadEvent.rcp to choose a different filename.

2.4 Built-in Configurations and Examples

The tsim_l1l2 package comes with some example code and a few pre-set configuration files. Several examples of Dataflow "spies" (routines which intercept information in the dataflow for package development and analyze output) are provided for developers and several controller rcp files are provided for users. The controller rcps include:

- head.rcp - runs all available L1 and L2 components
- head_analyze.rcp - runs all available analyze packages to fill a common ROOT file

These files can, of course, be customized by the user, they are merely provided as examples of tested, functional configurations.

2.5 Inter-Package Communication

2.5.1 The DataBroadcast Class

The DataBroadcast class is part of the tsim_l1l2 package. All data passed from level 1 to level 2 packages or passed between level 2 packages is sent to the offline framework inside a DataBroadcast. Consider the following example code used to pass an MBT channel from a level 1 processor to the framework:

```
 //
/ / put mbt channel into Databroadcast class
//
DataBroadcast *output=new DataBroadcast(4096);
output->store(mbt_track);
//
// put broadcast in offline framework datastore
//
produceItem(ds,output,l1example_id);
```

All level 1 and level 2 packages are required to communicate with level 2 via MBT channels stored in DataBroadcast classes in this manner. Passing data to the l1l2collector is described in a later chapter.

2.5.2 What about IOGEN? MBT Channels??

Though neither IOGEN nor MBT channel classes are part of tsim_l1l2, for those developing packages to run in tsim_l1l2 knowledge of these topics is essential.

IOGEN is a python package (l2iogen) which, given a configuration file, automatically generates C++ code to handle the IO of L2 packages. The classes it produces know how to pack data into a standard format which can be read back by other packages using the same classes. All L1/L2 packages in the simulation use IOGEN generated classes (included as part of the l2io package) to define the tracks, electrons, jets, etc. that they wish to pass to other packages. A complete IOGEN manual may be obtained from the L2 software web site (http://d0lxmsu1.fnal.gov/L2).

MBT channels define the standard L2 format for data movement. IOGEN objects are added to an MBT channel using its addobject method.When an IOGEN object is added to an MBT channel the MBT channel knows how to add (or update) the appropriate L2 header and L2 trailer information.

As an example of how this whole thing works: 25 IOGEN objects called "jets" are created in a given event by the L2 caljet processor, they are all added to a single MBT channel, this MBT channel is then wrapped in a DataBroadcast class and put into the Dataflow to be picked upby the L2 global prcessor.

3 Configuration via CoorSim

This is coor documentation. Sorry this does not exist yet.

4 Sending Data to L3 (l1l2collector)

The l1l2collector has the task of collecting all level 3 output from level 1 and level 2 packages and writing it to the RawDataChunk. By definition the collector is the last package to run in the Dataflow (it is forced to always be last using the finishReady method). It expects to receive data wrapped in L3CrateBroadcast classes which are very similar (and inherit from) the DataBroadcast class defined in the previous section but require a crate and module number be specified.The l1l2collector will attempt to collect all objects in the Dataflow which have been "aliased" as l3output in the produceItem call of a Dataflow package (ie. produceItem(datastore,item,name,l3output)).

5 Adding an L1 Processor

The simulation packages for the L1 trigger system (e.g., tsim_l1cal, tsim_l1ft) must present outputs to the L2 preprocessors in a way that mimics the online system. Inputs and outputs to the L2 system are all done throught the Data Flow portion of the offline framework (ref). Briefly, once all outputs are ready for transport, one packs that output into the data classes that L2 uses (which are created by the I/Ogen framework). These classes are wrapped up into FillableMBTChannels and MBTChannels, which are the classes that perform the same function as the cables and MBT cards do in the online system - they carry and hold the data from one system to the other. Theses classes themselves are presented to the rest of the simulation packages by putting them into the "data store". Once in the data store, any other package in the data flow framework can access the information. Please see the references for more information about the data store and the data flow framework.

In the first section, we outline the necessary functions in a DataFlow package, using the example of building a simulator for the L1CTT. The end of the first section contains a reasonably detailed example of actual header and source code. The second section delineates how to put your new package into the tsim_l1l2 package.

5.1 Being Part of the DataFlow

In order to be part of the DataFlow, a program must do 3 things:

- The class itself must inherit from DFPackage
- The class must have the public member functions:
  `fwk::Result ready(fwk::DataStore& ds),`
  `void outputs(fwk::DFPackage::StrList& l);`
  `void inputs(fwk::DFPackage::StrList& l)`
- The class must have private data members holding the id's for the data flow, e.g., `fwk::Action::Id l1cft1_id`. These will be filled in from RCP calls in the code.

5.1.1 The inputs Function

The inputs function provides access to the event from the DataStore. Typically, it will look like:

```
void L1CTT::inputs(fwk::DFPackage::StrList& l)
{
    l.push_back("event");
}
```

This allows the package to access the object on the DataStore called "event". We will discuss passing the event to the processEvent function in the section describing `ready`.

5.1.2 The outputs Function

The outputs functions puts the output classes onto the DataStore for access by other DataFlow packages. Typically, it looks like:

```
void L1CTT::outputs(fwk::DFPackage::StrList& l) {
  l.push_back(packageRCP().getString("l1ctt1")); //use the name from the RCPfile
  l.push_back(packageRCP().getString("l1ctt2"));
  l.push_back(packageRCP().getString("l1ctt3"));
  l.push_back(packageRCP().getString("l1ctt4"));
  l.push_back(packageRCP().getString("l1cttfrm"));
}
```

Note that the id used by the DataFlow framework is contained in your package's RCP file. Your RCP file will need to have lines like:

```
string l1ctt1 = “l1cft1”
```

Any package that wants to access these outputs just needs to pull `l1cft1` off the DataStore.

5.1.3 The ready function

The ready function takes the DataStore as it's argument, providing access to and from it. This function is the heart of the DataFlow framework, much like processEvent is the heart of Controller type of framework. In it, you call the processEvent function in your package and fill all the MBTChannel outputs to the L2.

5.2 Example: Toy Version of L1CTT

Here is an example of the header and source files for a toy L1CTT simulator. You can get most of this from the file ~casey/trigsim/t91/tsim_l1l2/src/L1CTT.cpp on d0mino. Note that processEvent needs to be filled in!
First, the header file:

```
// File: tsim_l1l2/L1CTT.cpp
// Purpose: Class to simulate output from a Level 1 example processor and
// pass into the Level 2 dataflow framework.
//
#include <string>
#include <iostream>
#include "framework/Registry.hpp"
#include "framework/Result.hpp"
#include "framework/Action.hpp"
#include "framework/DFPackage.hpp"
#include "edm/Event.hpp"
#include "d0om/d0_Ref.hpp"
#include "l2base/L2.hpp"
#include "l2base/io.hpp"
#include "l2io/FillableMBTChannel.hpp"
#include "tsim_l1l2/DataBroadcast.hpp"
#include DATAHEADER(L1CTTTrack)
#include IOHEADER(L1CTTTrack)
class L1CTT : public fwk::DFPackage {
public:
   L1CTT(fwk::Context*);
  ~L1CTT();
   fwk::Result ready(fwk::DataStore&);
   void inputs(fwk::DFPackage::StrList&);
   void outputs(fwk::DFPackage::StrList&);
   void processEvent(edm::Event &event); // where the work gets done
private:
   fwk::Action::Id l1cft1_id; // ids for the dataflow framework
   fwk::Action::Id l1cft2_id;
   fwk::Action::Id l1cft3_id;
   fwk::Action::Id l1cft4_id;
   cttTrack CTTTrackArray[100]; // Array holding the results from processEvent.
};
```

Now for the source code:

```
#include “l1ctt/l1ctt/hpp”
using namespace edm;
using namespace std;
using namespace fwk;
using namespace l2io;
using namespace tsim_l1l2;
//
// register this package with the framework
FWK_REGISTRY_IMPL(L1CTT,"$Name: $")
// Constructor
// All l1 simulation need to inherit from fwk:DFPackage ("DataFlowPackage")
// in the constructor.
L1CTT::L1CTT(fwk::Context* con): DFPackage(con){
  //
  // get id (label) of the outputs to the DataFlow for this processor. There
  // are 4 for the track info and 1 for the andor terms going to the l1fwk
  l1cft1_id=Action::instance()->mapNameToID(packageRCP().getString("l1ctt1"));
  l1cft2_id=Action::instance()->mapNameToID(packageRCP().getString("l1ctt2"));
  l1cft3_id=Action::instance()->mapNameToID(packageRCP().getString("l1ctt3"));
  l1cft4_id=Action::instance()->mapNameToID(packageRCP().getString("l1ctt4"));
  l1cftfrm_id=Action::instance()->mapNameToID(packageRCP().getString("l1cttfrm"));
}

//Destructor
L1CTT::~L1CTT() { // do the cleanup here }
// processEvent function
void processEvent(edm::Event &event){
   // Do all the stuff you've always done in the processEvent function
   // In particular, you will fill some object (e.g., CTTTrackArray) whose
   // contents will be sent to L2.
   cout << "In processEvent" << endl;
}
// ready function
// This is the function called by the Data Flow framework. It provides access
// to the event. The processEvent function is called from here.
Result L1CTT::ready(DataStore& ds) {
  cout << instanceName() << "/L1CTT::ready " << endl;
  // Get the event from the data store
  Event* e = fwk::getEvent(ds);
  if(e==0) { // check that there is an event!
    error_log(ELfatal,"ready") << "No Event found! Bad!" << endmsg;
    return Result::failure;
  }
  // Process the event
  processEvent(*e);
  //
  // Event processing is done. The information you want to send to L2 should
  // be contained in data members of the L1CTT class. Now, you will add this
  // information to the output classes to L2.

  cout << " L1CTT: adding elements " << endl;
  //
  // Create the FillableMBTChannels that will be put on the DataStore
  // The template arguments in each case are the IOgen data class for the
  // object and the maximum number of objects allowed. The initialization
  // arguments (e.g., CTT_CFT1,0,1) are the datatype (from
// l2base/datatypes.hpp), the major version, and minor version number of the
// IOgen output class.
//
FillableMBTChannel<L1CTTTrackData,15> mbt_track1(CTT_CFT1,0,1);
FillableMBTChannel<L1CTTTrackData,15> mbt_track2(CTT_CFT2,0,1);
FillableMBTChannel<L1CTTTrackData,15> mbt_track3(CTT_CFT3,0,1);
FillableMBTChannel<L1CTTTrackData,15> mbt_track4(CTT_CFT4,0,1);
// Create an instance of the IOgen class that will be filled and added to
// the FillableMBTChannel (and then onto the DataStore).
L1CTTTrackData cttTrack;

// Now, loop over all the tracks, sending them to the appropriate outputs.
for(int itrack=0; itrack<Ntrack; itrack++){
  // Set the data members of cttTrack. You can see what the datamembers are
  // by looking at the l2io/l2io.iogen file. I've shown all that are there
  // as of t92, which I got from the TDR. It is important to remember that
  // since the IOgen classes do all the packing, you must get the data types
  // correct. For instance, all of the unpacked data members to the
  // L1CTTTrackData class are uint32 objects (defined in
  // l2base/types.hpp), therefore, all of the accessors in the local array
  // CTTTrackArray must return uint32 quantities.
  cttTrack.setIsolation(CTTTrackArray.iso());
  cttTrack.setPSThresh(CTTTrackArray.psThresh());
  cttTrack.setErrorCode(CTTTrackArray.errCode());
  cttTrack.setPT(CTTTrackArray.pt());
  cttTrack.setPTbin(CTTTrackArray.ptbin());
  cttTrack.setSign(CTTTrackArray.sign());
  cttTrack.setSector(CTTTrackArray.sector());
  cttTrack.setFibreNumber(CTTTrackArray.fiber());
  //
  // Put track into the appropriate MBT Channel, which are assigned based upon
  // sectors. (I think this is how it works, but I'm really making it up.)
  if(CTTTrackArray.sector() == 1) mbt_track1.addObject(cttTrack);
  if(CTTTrackArray.sector() == 2) mbt_track2.addObject(cttTrack);
  if(CTTTrackArray.sector() == 3) mbt_track3.addObject(cttTrack);
  if(CTTTrackArray.sector() == 4) mbt_track4.addObject(cttTrack);
}
//
// Now, all the tracks are filled, but the output to the L1 framework is left
// Create the output to the L1FRM
L1FRMAndOrData *terms=new L1FRMAndOrData;
terms->setSourceID(1);
terms->setNActive(64);
bool andor1=true;
int andor_id1=25;
for (int i=0;i<64;i++) {
      int j = i+1;
      AndOrPair cal_andor1(bits[i],j); // AndOrPair is defined in tsim_l1fwk
      terms->setAndOr(i,cal_andor1.get_cable());
    }
    //
    // Now, all the MBTChannels are filled. All that is left is to put them in
    // the DataStore to be sent to L2.
    // For the L2 input/output, we use the DataBroadcast class to hold the data.
    // We need one for each output to L2, i.e., one for each MBTChannel
    //
    DataBroadcast *output1=new DataBroadcast(4096);
    DataBroadcast *output2=new DataBroadcast(4096);
    DataBroadcast *output3=new DataBroadcast(4096);
    DataBroadcast *output4=new DataBroadcast(4096);
    // Put the FillableMBTChannels into the DataBroadcast objects
    output1->store(mbt_track1);
    output2->store(mbt_track2);
    output3->store(mbt_track3);
    output4->store(mbt_track4);
    //
    // Put DataBroadcast objects in the datastore
    produceItem(ds,output1,l1cft1_id);
    produceItem(ds,output2,l1cft2_id);
    produceItem(ds,output3,l1cft3_id);
    produceItem(ds,output4,l1cft4_id);
    produceItem(ds,terms,l1cftfrm_id);
    cout << "exiting l1ctt " << endl;
    return Result::success;
  }
  //
  // inputs method tells framework which inputs are required for
  // ready method to run.
  void L1CTT::inputs(fwk::DFPackage::StrList& l) {
    l.push_back("event"); // We want to be able to access the event!
  }
  //
  // outputs method tells framework which outputs are produced by ready method
  void L1CTT::outputs(fwk::DFPackage::StrList& l) {
    l.push_back(packageRCP().getString("l1ctt1")); //use the name from the RCPfile
    l.push_back(packageRCP().getString("l1ctt2"));
    l.push_back(packageRCP().getString("l1ctt3"));
    l.push_back(packageRCP().getString("l1ctt4"));
    l.push_back(packageRCP().getString("l1cttfrm"));
  }
```

5.3 Being part of tsim_l1l2

Here is a "play-by-play" action list for taking a header file and source file (l1ctt.hpp and l1ctt.cpp above), and creating a package and making the appropriate changes to tsim_l1l2 in order to run it.

- Setup your release area (called `myRel` from now on) and change directories to it.
- Create a new package in your release area (called `l1cttsim` from now on), and create the link to the package header directory in the `myRel/include` directory.
- Put your header and source file in the header and source directories of `l1cttsim`. Don't forget to put DOC++ comments in the header files. Also, be generous with your comments in general - other folks have to read your code!
- Add `l2base` and `l2io` to `l1cttsim/LIBDEPS`
- Add the source file to `l1cttsim/src/COMPONENTS`
- Add a dummy test file for your source code to the `l1cttsim/src` directory (e.g., `%> echo "int main(){return 0;}" >> l1ctt_t.cpp`). This is so that your code doesn't fail the component test in the build system.
- Add a framework initialization file –`l1cttsim/src/`fwkL1CTTSIM.cpp:
  ```
  #include "framework/Registry.hpp"
  FWK_REGISTRY_DECL(l1cttsim)
  ```
  and add that file to the `OBJECT_COMPONENTS` file in l1cttsim/src (e.g., `%> echo fwkL1CTT >> OBJECT_COMPONENTS`)
- Create an RCP file for your package (`l1cttsim.rcp`) and put it in `l1cttsim/rcp`. It should look like:
  ```
  string PackageName = "l1cttsim"
  // here is the stuff for the MBT outputs
  string l1ctt1 = "l1cft1"
  string l1ctt2 = "l1cft2"
  string l1ctt3 = "l1cft3"
  string l1ctt4 = "l1cft4"
  string l1cttfrm = "l1cftfrm"
  ```

Now, your package is ready to be added to tsim_l1l2.

- Add the tsim_l1l2 package to your release area, i.e., `%> addpkg tsim_l1l2`, which is just add the current release version to your local area.
- Add the name of the package (`l1cttsim`) to the `tsim_l1l2/bin/LIBRARIES` file.

- Add the name of the framework registration file (fwkL1CTTSIM) to the `tsim_l1l2/bin/OBJECTS` file.

- Create a new "head" rcp file for running `tsim_l1l2` by modifying `tsim_l1l2/rcp/head.rcp` and `tsim_l1l2/rcp/tsim.rcp`. We'll call the new files `myhead.rcp` and `mytsim.rcp`. I've highlighted the changes in *italics*.
  First `myhead.rcp`:

  ```
  string InterfaceName = "process"
  string Interfaces = "generator l2configure process dump"
  string Flow = "generator l2configure process dump"
  string PackageName = "Controller"
  string Packages = "read calfe cttfe runConfigMgr configure tsim dump"
  RCP read = <tsim_l1l2 ReadEvent>
  RCP calfe = <calunpdata CalMCToUnp>
  RCP cttfe = <whatever it is>
  RCP runConfigMgr = < run_config_mgr run_config_mgr >
  RCP configure = <tsim_coor L2CoorMgr>
  RCP tsim = <tsim_l1l2 mytsim>
  RCP dump = < tsim_l1l2 dumpevent >
  ```

  I don't know what package produces the front-end information for the l1ctt. The point is, however, that whatever packages that are not part of the dataflow (for instance, you need to run a package that creates some unpacked data chunks which you will read in later – this is what tsim_l1cal does) tha need to run before your new package should be run in the rcp file above.
  Now, `mytsim.rcp`:

  ```
  string PackageName = "Dataflow"
  //string Packages = ( l1cal l1frm l2calem l2caljet l2calmet global collector)
  string Packages = ( l1cal l1ctt)
  RCP l1cal = < tsim_l1cal tsim_l1cal >
  RCP l1ctt = < l1cttsim l1cttsim >
  RCP l1frm = <tsim_l1l2 tsim_l1frm>
  RCP l2calem = < tsim_l1l2 l2_calem >
  RCP l2caljet = < tsim_l1l2 l2_caljet >
  RCP l2calmet = < tsim_l1l2 l2_calmet >
  RCP global = < tsim_l1l2 l2_global >
  RCP collector = < tsim_l1l2 l1l2_collector >
  ```

  Commenting out the entire data flow (the Packages string) allows you to just run the the portions of the the simulation that you are interested in.

Now, you should be able to recompile and try running `tsim_l1l2` by going into the `tsim_l1l2/bin` directory and typing `./Runme.sh`

5.4 Places to look for help

We are still putting together documentation. If you have questions, please feel free to ask Dylan Casey (casey@pa.msu.edu), Dugan O'Neil (oneil@fnal.gov), Roger Moore (moore@pa.msu.edu), or Josh Kalk (kalk@pa.msu.edu). Dylan and Josh put tsim_l1cal into the DataFlow model and made it output to the L2 preprocessors. The tsim_l1cal package is also useful to look at as an example of a working package.

6 The L1 Framework (l1frm)

This is a brief description of what the framework does, what it requires as input, etc.

7 Adding an L2 Worker

Adding an L2 worker to the simulation is somewhat different than adding an L1 worker. L2 worker code is online code and therefore must conform to online coding guidlines. A standard offline interface to the online worker code has been provided as part of the tsim_l2 package. Simulation inputs and outputs are created and managed by this interface. This means that the worker code itself must only create and fill its IOGEN-generated objects. The filling of MBT channels, creation of DataBroadcast classes and sending of information to the Dataflow dispatcher is all handled by the interface.

In order to more easily define the input and output formats expected by the online worker a tool called EVIGEN has been provided as part of the l2tools package. This is a python package which parses and input text file and generates C++ classes which can be used to define the formats. The inputs are presented to the worker by the offline interface. The output definition is a defintion of the L3 output format of the worker. From this the offline interface can create the L3 output and can extract the L2 global output and send these items to the Dataflow Dispatcher. An example EVIGEN output configuration is given below:

```
  [IOBlock]
package = l2caljetworker
worker = CalJet
type = output
blockID = CALJET_STD_OUTPUT
majorver = 0
minorver = 1
incdir = %(package)s
srcdir = src
l2output = jet
[Channel0]
```

```
name = jet
object = CALJet
type = unpacked
limit = 50
sourceID = GBL_JET
stdfetch = l2workerbase/fetchFromStore
ubsfetch = l2workerbase/fetchFromStore
majorver = 0
minorver = 1
comment = Output jets from L2CALJet
```

No EVIGEN manual exists. To write this file for your package you will need help from the authors of this manual.

Once the inputs and outputs for the package have been defined through EVIGEN and the worker code has been written two packages need to be made away of the existence of the new worker: tsim_l2 and tsim_l1l2. To add the worker to tsim_l2 a simple routine to instantiate a worker of this new type must be written and added to tsim_l2/src/. An example of such a file is (L2CalJet.cpp for the calorimeter jet worker):

```
#include <iostream>
#include "framework/Registry.hpp"
#include "tsim_l2/L2Interface.hpp"
#include "l2caljetworker/CalJetWorker.hpp"
#include "l2caljetworker/CalJetInput.hpp"
#include "l2caljetworker/CalJetOutput.hpp"
using namespace fwk;
using namespace tsim_l2;
using namespace l2caljetworker;
typedef class L2Interface<CalJetWorker> L2CalJet;
FWK_REGISTRY_IMPL(L2CalJet,"$Name: $")
```

...this is the whole file. Then add this filename to the tsim_l2 COMPONENTS file. Next include a line like:

```
 FWK_REGISTRY_DECL(L2CalJet)
```

in fwkRegTsiml2.cpp and include your new worker library in its LIBDEPS file and tsim_l2 is ready. To add the packjage to tsim_l1l2 it is only necessary to add your rcp file to tsim_l1l2/rcp and reconfigure the tsim_l1l2 Dataflow controller (eg. tsim.rcp) to run your worker.

8 Global Tools and Filters

global documentation. Includes list of available tools. References Dylan's document on how to write a tool/filter.

References

[1] This is the Kowalkowski Framework user's guide